Growing Triples on Trees: an XML-RDF Hybrid Model for Annotated Documents

François Goasdoué, Konstantinos Karanasos, Yannis Katsis, Julien Leblay, Ioana Manolescu, Stamatis Zampetakis

Growing Triples on Trees: an XML-RDF Hybrid Model for Annotated Documents *

François Goasdoué[1] Konstantinos Karanasos[1] Yannis Katsis[2]
Julien Leblay[1] Ioana Manolescu[1] Stamatis Zampetakis[1,3]

[1]Leo team, INRIA Saclay and LRI, Université Paris-Sud 11
[2]INRIA Saclay and ENS Cachan
[3]University of Crete
firstname.lastname@inria.fr

Abstract

Une partie grandissante des données produites et exploitées de nos jours sont structurées sous la forme de documents, typiquement XML. Dans de nombreux domaines d'application (données scientifiques, données des réseaux sociaux, blogs, entrepôts thématiques de documents...), nous voyons émerger le concept de documents avec annotations, permettant d'enrichir le sens d'un document produit par un auteur ou une organisation avec des informations supplémentaires provenant d'autres sources.

Dans cet article, nous présentons XR, une approche unifiée et inspirée des standards du Web (XML, RDF, et RDFS) pour décrire des documents et des annotations. Le concept central dans XR est que chaque nœud XML est une ressource qui peut être annotée par des triplets RDF. Nous définissons le modèle de données XR, son langage de requête XRQ et sa sémantique. Nous avons développé XRP, un premier prototype de gestion de données XR, et démontrons sa faisabilité par une série d'expériences.

1 Introduction

With its widespread acceptance, XML has become the language of choice for publishing semistructured data on the web, be it scientific (e.g., SwissProt), governmental (e.g., www.data.gov) or business data.

In parallel, RDF is becoming the de facto standard for describing semantically rich data. Its provision (combined with RDFS) for defining classes and properties, as well as relationships between them (subsumption relations between classes or properties, and typing of properties' attributes), which are then taken into account to answer queries, make it ideal for use in such situations.

Until now these two data models have led separate lives (up to some work on translating one to the other discussed in Section 2). However, we argue that this does not

*This work has been partially funded by *Agence Nationale de la Recherche*, decision ANR-08-DEFIS-004, and by the FP7 European Research Council grant agreement Webdam number 226513.

have to be this way: By combining the power of both models, we could enable many real-life scenarios that require XML data annotated with semantics (which in turn need to be taken into account in query answering).

Consider the following scenario: A newspaper is publishing its articles in XML format. To allow users to easier query its articles, it decides to classify them according to their topics w.r.t. a publicly available ontology on news articles. To store classification information, the administrator needs to be able to assign each article to (potentially multiple) concepts of the news ontology. Moreover, she needs an automatic mechanism to reason on the ontology, so that whenever a user searches for article topics (e.g., biology), she also gets back articles on more specific ones (e.g., bioengineering, biofuel energy etc.). While XML does not make a provision for this scenario, RDF could be used to link the article node to the corresponding ontology concepts. Furthermore, if both XML and RDF could be combined, the user would be able to query the document both on semantic relationships as well as on structural relationships (from the XML document).

This work aims to enable such scenarios, by proposing a unified model allowing the combination of XML data with RDF data into a single instance. We have designed the model and the corresponding query language and implemented a system for storing and querying instances of the proposed model. Moreover, we showcase optimizations that are possible when XML and RDF are combined in the same instance. This work makes the following contributions:

Data Model for Annotated XML Documents In contrast to most existing works that allow only the representation of either XML data or RDF data or the union thereof, the proposed data model can even express instances where XML and RDF are actually interconnected (e.g., when an RDF triple refers to an XML node).

Query Language Unlike existing approaches where (as part of the query) XML data has to be converted first to RDF before being joined with the second (or vice versa), our query language allows writing queries that filter both according to the structural and the semantic relationships, without having to first convert data from one format to another.

Implementation & Optimizations In contrast to existing systems, where RDF data have to be translated to XML before being queried (or vice versa), the proposed system allows keeping the XML and RDF data in two separate physical stores. However, at query processing it employs a single query evaluation engine that treats both XML and RDF data uniformly, allowing for cross-model optimizations that would not be possible if XML and RDF were queried by separate query engines.

Experimental Results Our experiments show the behavior of the system for different types of queries and the effectiveness of the employed optimizations.

The paper is structured as follows: We start by discussing related work in Section 2. Section 3 presents representative use cases and the running example, used throughout the paper. Subsequently, Sections 4 and 5 describe the data model and query language, respectively. The implemented system is discussed in Section 6, followed by the experimental results in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

Two major lines of work are closely related to this paper: The first shares our motivation of annotating structured data and the second is related to our technical results of combining XML with RDF. We start by discussing works on annotations.

Tools for annotating web pages As RDF emerged, a lot of works, proposed frameworks that let users semantically annotate web-pages either in a manual [37, 19] or in a (semi-)automatic fashion [27, 12] (a comprehensive overview of annotation systems can be found in [25]). However, these works focus solely on the storage and querying of annotations and do not consider the problem of querying simultaneously data structures and the annotations on them.

The WebContent R&D project [1] is one among several recent efforts of building thematic warehouses of Web data, and in particular crawled pages from the Web, processed with natural language analysis tools to identify named entities and label the documents according to some given ontologies. Our data model extends the WebContent data model by allowing XML nodes to be referenced in RDF in all places where a URI can appear (as opposed to only subjects in WebContent), enabling much richer data instances.

Embedding RDF annotations in XML documents Similar to these works but much more focused on how to publish RDF annotations in XML documents are the recent recommendations for embedding RDF annotations in XHTML: microformat [30], eRDF [28] and W3C's RDFa [34] standard. Nevertheless, they do not cover the querying aspect and they assume users are allowed to modify the original XML documents to add annotations. This is particularly restrictive, especially when users want to keep their annotations private, or when they do not have access rights to the documents. In contrast, we aim for a system where RDF annotations do not have to be embedded in the XML documents, but can instead be kept in a separate physical store.

Parallel to the work on annotations, a significant amount of work has been spent on studying the connection between RDF and XML.

From RDF to XML and back In this context, several works propose languages that allow, as described in W3C's GRDDL recommendation [29], the transformation of XML data to RDF and vice versa [3, 9], known in the literature as *lifting* and *lowering*, respectively. Some look into employing a query language of one model to query the other (e.g., using XQuery to query RDF) [26, 24, 13] or building hybrid languages that embed constructs of a query language for one model (e.g., XPath) into a query language for the other model (e.g., SPARQL) [11]. Finally, some works present general frameworks that allow the modeling of different query languages [15].

Note that some of these approaches solve the task of querying combined XML and RDF data. Generally, they accomplish it by transforming RDF to XML data and then using XQuery to query the original XML documents together with the XML-ized RDF triples. This is also the approach we followed in our recent work [20]. The converse, that is, transforming XML to RDF and querying both with SPARQL, has also been considered. However, (a) writing such queries can be cumbersome, (b) the queries incur the cost of transforming data to some other format first, and (c) these approaches do not consider the particular case where RDF triples refer to XML nodes. Instead, we propose a system where queries can transparently combine XML and RDF without having to convert or alter existing data.

3 Use Cases

The ever increasing volume of documents published by various organizations is making their efficient manipulation progressively harder. Users are turning to annotations as a means of enriching such data with semantic information, which can subsequently aid in its manipulation. In this Section, we present various scenarios in which annotations are becoming essential.

A prominent use case is annotation of commercial information, such as in the newspaper scenario presented in the introduction. Newswires, financial quotes, weather reports and other data feeds are annotated through various channels: manually by publishers, journalists and end-users, or automatically by extracting RDF from XHTML documents [29], or via NLP-based services such as OpenCalais[1] to disambiguate words and extract named-entities. Annotations are added at different granularities (from single words to bigger chunks of text and entire documents) and made available to end-users to more efficiently filter through the magnitude of available information.

A similar emergence of annotations can be witnessed in social networks. Social networks and most recently even search engines are providing ways to let users reward contents they like, and express their feelings. Meanwhile, methods are devised to automatically extract such opinions from blogs, forums, product reviews, etc. Sentiment mining is gaining momentum in the data analysis communities. Storing this type of information as semantic annotations would enable classifying opinions into hierarchical structures. A wide range of applications could benefit from a data model that combines XML with RDF, from crisis management to market surveys.

Last but not least, annotations play a prevalent role in Health Sciences and, in particular, in the fields of Systems Biology and Bioinformatics. For instance, UniProt[2] is a protein database containing annotations describing the function of each protein, its domain structure, etc. Another example is BLAST[3], in which annotations are used to define similarities between parts of different DNA or protein sequences. Many of these databases were initially modeled in XML, and were lately transformed to RDF to exploit the new capabilities its model provides. However, structural relationships cannot be rendered as efficiently via pure RDF. Thus, a unified approach featuring the best of both data models would be beneficial in this context.

Running example To illustrate the proposed concepts, we employ a product catalog scenario: Consider a web-site that integrates information about products from various sources, such as product feeds of on-line retailers (containing information about the products they sell) and the HTML pages of amazon.com containing product descriptions and reviews. The data is integrated using automatic inference techniques that create annotations over it, e.g., linking an item in the product feed to its corresponding description on amazon.com. On top of that, users can manually create two types of annotations: To aid search, they can annotate a product with its type using a predefined ontology, moreover, they can add social networking annotations to promote products they like and make connections to other users in the system. We will present a concrete example of such data in the following section.

4 The XR Data Model

To represent annotated documents, we introduce the *XR data model*. In keeping with the widely accepted standards for representing semi-structured data (i.e., XML) and semantic

[1] http://www.opencalais.com/
[2] http://www.uniprot.org/
[3] http://blast.ncbi.nlm.nih.gov/

relationships (i.e., RDF), an instance of the XR data model comprises two sub-instances: An XML sub-instance, consisting of a set of XML trees, and an RDF one, consisting of a set of RDF triples. The connection between the two is achieved by assigning to each XML node a unique Uniform Resource Identifier (URI), which can then be referred to from an RDF triple, as we will explain below.

Next, we formally define XR sub-instances. We rely on a set $\mathcal{U}$ of URIs as defined in [35], and a subset $\mathcal{I} \subseteq \mathcal{U}$ of *document* identifiers or, equivalently, document URIs. We denote by $\mathcal{L}$ the set of literals [32] (which for simplicity can be seen as the set of all strings). $\mathcal{N}$ is the set of possible XML element and attribute names, to which we add the empty name ϵ. Finally, $\mathcal{B}$ is a set of blank nodes (accounting for unknown literals or URIs, as we will explain later on). An XML tree is defined as usual:

Definition 4.1 (XML Tree) *An* XML tree *is a finite, unranked, unordered* [4]*, labeled tree $T = (N, E)$ with nodes N and edges E, where each node $n \in N$ is assigned a label $\lambda(n) \in \mathcal{N}$ and a type $\tau(n) \in \{document, attribute, element, text\}$. An attribute node must be the child of an element node, it has a value belonging to $\mathcal{L}$ and it does not have any children. A text node can only appear as a leaf. Finally, an XML tree can have at most one document node. The document node can only appear as the root of the tree, has exactly one child and has the empty name ϵ.*

Most frequently, we are concerned with trees that are also documents, i.e., those rooted in document nodes. However, we may also consider trees rooted at simple XML elements, for instance, when XML trees are passed from the output of one query to the input of another, without being permanently stored within a document. In our examples we will consider only XML documents and thus omit the document node at the root of the tree. A set of XML trees forms an XML instance:

Definition 4.2 (XML Instance) *An* XML instance *I_X is a finite set of XML trees.*

We assume available a function that assigns a unique URI to each node in an XML instance. A URI of a document node is commonly referred to as the *document URI*.

The URI assignment function is crucial for interconnecting the XML and RDF sub-instances, as we will explain below: The unique identifiers assigned to the nodes allow the RDF sub-instance to refer to nodes of the XML sub-instance. While discussing our system implementation in Section 6, we present such a URI assignment function that can be used in practice. However, for the purpose of the definitions, it suffices to consider any URI assignment function acting like a Skolem function, i.e., returning a new ("fresh") value every time it is called for the first time with a given input, and consistently returning that value to any subsequent call with the same input.

To facilitate the connection between the RDF and the XML sub-instance, an RDF sub-instance is defined as a set of triples, which can among others refer to the URIs of the XML nodes:

Definition 4.3 (RDF Instance) *An* RDF instance *I_R is a set of triples of the form (s, p, o), where $s \in (\mathcal{U} \cup \mathcal{B})$, $p \in \mathcal{U}$, and $o \in (\mathcal{L} \cup \mathcal{U} \cup \mathcal{B})$.*

[4]It is trivial to extend the definition to partially ordered trees, as per the XML standard.

Following the common nomenclature, the components of a triple (s, p, o) are referred to from left to right as its *subject*, *property* and *object*, respectively.

As defined above, the subject or the object of the triple can be bound to a so-called *blank node*. Blank nodes are used in RDF [31] to denote *unknown URIs or literals*, similarly to *labelled nulls* in the database literature [2]. For instance, one can use a blank node b_1 in the triple $(b_1, country, \text{“}France\text{”})$ to state that the *country* of b_1 is *France*, without using a concrete URI. Blank nodes can be repeated in an RDF instance, thus allowing multiple triples to refer to the same unknown URI or literal. For example, a second triple $(b_1, city, \text{“}Paris\text{”})$ could specify that the *city* of the same b_1 is *Paris*. Finally, multiple blank nodes can co-exist in a data set, thus allowing the representation of several unknown URIs or literals. For example, one may also state that the *country* of some other unknown URI b_2 is *Morocco*, while its *population* is an unspecified literal b_3.

Another peculiarity of RDF is that it does not model only explicit triples, but also implicit (a.k.a. *entailed*) triples. The latter can be derived from the former according to a set of *entailment rules*. More details on this process, known as *RDF entailment*, can be found in [33]; for the purposes of our discussion though, it suffices to be aware of the following: Given an RDF instance I_R, its semantics is the RDF instance I_R^∞, called the *saturation of* I_R, consisting of I_R plus all the implicit triples derived from I_R through RDF entailment. RDF entailment will be central for RDF (thus XR) query answering (discussed in Section 5.2), as answers must also take into account the implicit triples.

We can now define an XR instance as follows:

Definition 4.4 (XR Instance) *An* XR instance *is a pair (I_X, I_R), where I_X and I_R are an XML and an RDF data instance, respectively, built upon the same set of URIs.*

It is important to note that the XML and the RDF sub-instances are defined over the same set of URIs $\mathcal{U}$, thus allowing RDF triples to annotate nodes of XML trees. The following example illustrates such an interconnected XR instance.

Example. Figure 1 shows a sample XR instance corresponding to our running example of annotated product information. It consists of three XML trees linked through RDF annotations: the product catalog of a retailer split into two XML trees containing product and company information and an XHTML page from amazon.com containing product information. The RDF sub-instance is shown on the top part of the Figure while the three XML trees forming the XML sub-instance are shown on the bottom. For simplicity, we omit the document node at the root of each XML document (the same holds for the queries presented in the following section). The subscript next to the label of each XML node corresponds to its URI. URIs are used to allow the RDF triples to annotate the XML trees. For instance, the first triple specifies that Alice likes the product corresponding to the node with URI=“#12”. RDF triples can also link nodes across two XML trees. For example, the second triple specifies that the node with URI=“#12” corresponds to the description of the product represented by the node with URI=“#3”. Finally, the example also illustrates the two concepts we discussed about RDF: blank nodes and entailment. The fourth and fifth triples (shown on the second line of the RDF sub-instance) contain the blank node B; we do not know the identity of B, although we know his e-mail address and the fact that Alice knows him. Moreover, not all tuples shown in the RDF instance are explicit. The last triple (shown in gray) specifying that the product with URI=“#3” is an electronic device is implicit. It was derived from the fact that it is a mobile phone and that a mobile phone is an electronic device.

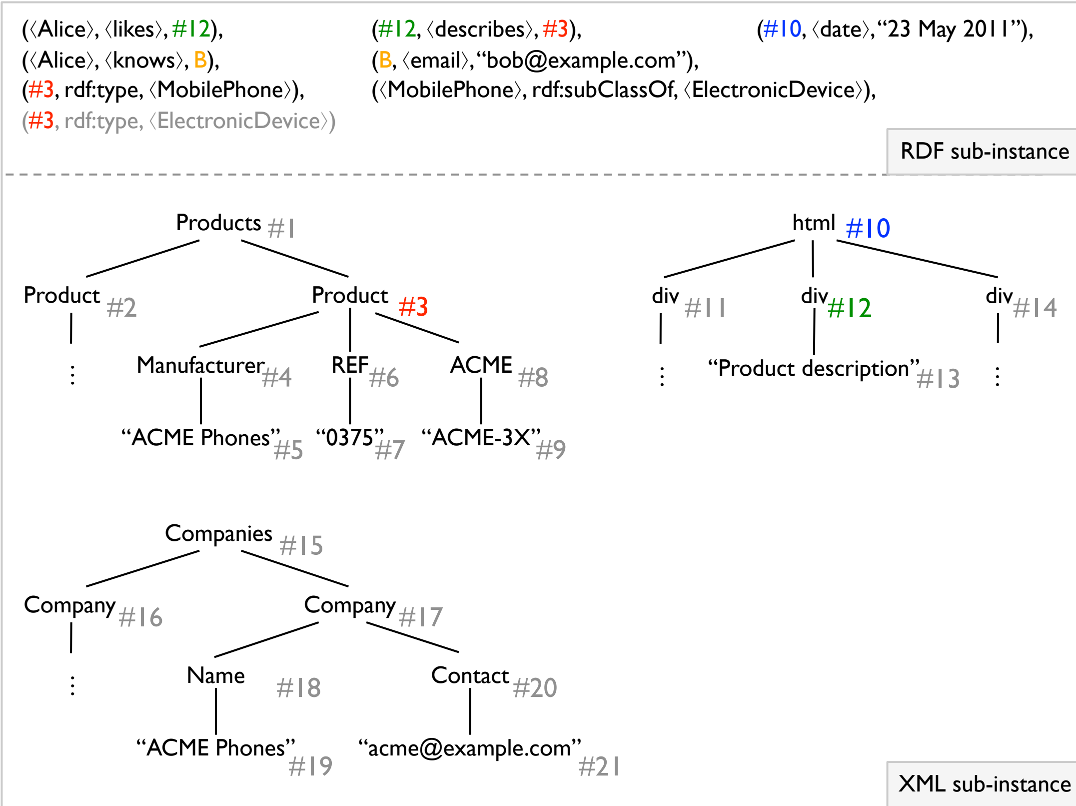


Figure 1: XR instance representing annotated product information

5 The XRQ Query Language

Given an XR data instance, users should be able to query the data based on both its structure (described in the XML sub-instance) and its semantic annotations (stored in the RDF sub-instance). To this end, we design XRQ: a query language that allows querying according to both axes. We start by defining XRQ in Section 5.1, before formally presenting its semantics in Section 5.2. Finally, in Section 5.3 we present an extension of our query language that allows construction of more complex output than the standard XRQ discussed below.

5.1 XRQ Definition

Staying close to the data model, XRQ consists of two main constructs: *tree patterns*, that allow filtering based on the XML sub-instance and *triple patterns* that allow querying based on the RDF sub-instance. Both types of patterns are defined below. Importantly, variables appearing in tree patterns can be reused in triple patterns, thus allowing queries to select data based on both their structure and their semantic annotations.

Definition 5.1 (Tree Pattern) *A tree pattern is a finite, ordered, unranked, $\mathcal{N}$-labelled tree with two types of edges, namely child and descendant edges. We may attach to each node at most one* uri *variable, one* val *variable and one* cont *variable. We may also attach to a node an equality predicate of the form* [val=c] *for some* $c \in \mathcal{L}$.

A tree pattern is a variant of tree patterns as presented in the literature [4] with the additional capability of attaching (one or more) variables of different types to the

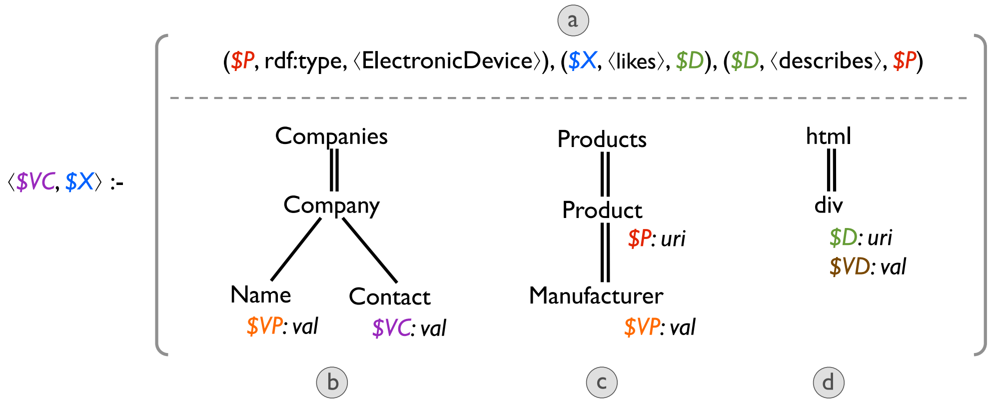


Figure 2: Sample XRQ query

nodes. Variables serve two purposes: (i) to denote data items that are returned by the query (in the style of distinguished variables in conjunctive queries) and (ii) to express joins between tree (or triple) patterns. The variable type specifies the exact information item from an XML node, to which the variable will be bound. When a node n_t of a tree pattern is matched against a node n_d of an XML tree, the variables attached to the node n_t will be bound to the following concepts, according to the variable's type: *uri* variables are bound to the URI of n_d. If n_d is an element, *val* variables are bound to the concatenation of all text descendants of n_d; if n_d is an attribute, *val* variables are bound to the attribute value. Finally, *cont* variables are bound to the serialization of the subtree rooted at n_d. The semantics of *val* variables are copied from the XPath (and XQuery) specification. Indeed, an XPath snippet of the form $x="Paris", where $x is bound to some XML element, is interpreted as: check if the concatenation of all text descendants of that element equals "Paris". We represent such predicates by annotating a tree pattern node with [*val*="Paris"]. Similarly, a comparison of the form ... where $x=$y ... is interpreted as: the value of $x (as we defined it above) is equal to the value of $y. Our queries also allow expressing such comparisons, as we will explain later on.

Example. The bottom part of Figure 2 shows graphically three tree patterns for our running example. As usual, single (double) edges correspond to parent-child (ancestor-descendant, resp.) relationships. For instance, the tree pattern on the right looks for an *html* node with a descendant *div* node. For each match of the pattern against the tree, $D will be bound to the URI of the matched *div* node, while $VD will be bound to the concatenation of all its text descendants.

Definition 5.2 (Triple Pattern) *A triple pattern is a triple* (s, p, o)*, where* s, p *are URIs or variables, whereas* o *is a URI, a literal, or a variable.*

Example. The top part of Figure 2 depicts three triple patterns. For instance, the left-most triple pattern asks for all URIs of type *ElectronicDevice*.

By combining tree and triple patterns and endowing them with a set of projected (head) variables, we obtain an XRQ query:

Definition 5.3 (XRQ Query) *An XRQ query consists of a* head *and a* body*. The body is a set of tree and/or triple patterns built over the same set of variables, whereas the head is a list of variables appearing also in the body.*

Note that by using variables in multiple places within the query, one can express joins. In general, three types of joins are possible: Joins between tree patterns, joins between triple patterns or joins between tree patterns and triple patterns. This property of XRQ facilitates queries that cross the boundaries between the XML documents and their RDF annotations. The following example illustrates the expressivity of XRQ.

Example. Figure 2 shows an XRQ query, whose body (shown on the right) comprises three triple patterns (shown on the top) and three tree patterns (shown at the bottom). It is asking for all products (second tree pattern) of type *ElectronicDevice* (first triple pattern), the companies that make this product (first tree pattern) and the corresponding product descriptions (last triple pattern and tree pattern), such that somebody has expressed that he likes this description (second triple pattern). In turn, it is returning the text value of the manufacturer's contact address together with that of the URI of the persons who expressed their satisfaction, as evidenced by the existence of the variables $VC and $X in the head of the query (shown on the left side of the Figure). Note the use of variables for expressing joins. The particular query showcases all possible types of joins: Joins between two tree patterns (through variable $VP), between two triple patterns (through variable $D) and between a tree pattern and a triple pattern (through variable $P). Color-coding is meant to assist the reader in finding all occurrences of a variable in the query. Note that the encircled letters do not form part of the query; they will be used for explanation purposes when we discuss the query semantics in the next subsection.

5.2 XRQ Semantics

We now formally define the semantics of XRQ. To this end, we first define the notion of *matches* and *variable bindings* for each of its components (i.e., tree patterns and triple patterns). A match of a tree pattern against an XML instance is defined as usual through tree embeddings [4]:

Definition 5.4 (Match of a tree pattern against an XML instance) *Let Q be a tree pattern and I_X an XML instance. A* match *of Q against I_X is a mapping ϕ from the nodes of Q to the nodes of I_X that preserves (i) node labels, i.e., for every $n \in Q$, $\phi(n) \in I_X$ has the same label as n and (ii) structural relationships, that is: if n_1 is a /-child of n_2 in Q, then $\phi(n_1)$ is a child of $\phi(n_2)$, while if n_1 is a //-child of n_2, then $\phi(n_1)$ must be a descendant of $\phi(n_2)$.*

A match of a tree pattern Q against an XML instance I_X defines the mapping of nodes of Q to nodes of I_X. However, recall that a tree pattern, apart from nodes, contains also variables, which have to be bound to objects. This mapping of variables to objects, referred to as a *variable binding* is formally defined below:

Definition 5.5 (Variable binding of a tree pattern against an XML instance) *Let ϕ be a match of a tree pattern Q against an XML instance I_X and V the set of variables in Q. Let $v \in V$ be a variable associated to a node n. Then the* variable binding *f of Q against I_X corresponding to ϕ is a function over V such that: (i) if v is a* uri *variable, then $f(v)$ is the URI of $\phi(n)$ in I_X, (ii) if v is a* val *variable, then $f(v)$ is the value of $\phi(n) \in I_X$, as we defined it in Section 5.1 and (iii) if v is a* cont *variable, then $f(v)$ is serialization of the subtree of I_X rooted at $\phi(n)$.*

As explained above, a variable binding f of Q against I_X is associated to a match ϕ of Q against I_X. For simplicity however, in the following we will assume the existence of a match and refer to f simply as a variable binding of Q against I_X.

To capture the semantics of the triple patterns, we define matches and variable bindings for them as well:

Definition 5.6 (Match of a triple pattern against an RDF instance) *Let Q be a triple pattern (s,p,o), I_R an RDF instance and I_R^∞ the saturation of I_R. A* match *of Q against I_R is a mapping from the triple (s,p,o) to some triple $t_\phi = (s_\phi, p_\phi, o_\phi) \in I_R^\infty$, such that $\phi(s) = s_\phi$, $\phi(p) = p_\phi$ and $\phi(o) = o_\phi$, and for any URI or literal $ul \in \{s,p,o\}$, $\phi(ul) = ul$ (ϕ maps any URI or literal only to itself).*

It is important to note that in accordance with the RDF semantics as specified by the W3C, a triple pattern is matched not against an RDF instance I_R, but against the saturation of I_R, denoted I_R^∞. As defined in Section 4, I_R^∞ contains in addition to the explicit triples of I_R, a set of implicit triples.

We recall the notion of *restriction of a function to a subset of its domain*. Let f be a function over a set A. The restriction of f to a subdomain $A' \subseteq A$, denoted by $f|_{A'}$, is a function f' over A', s.t. $f'(x) = f(x), \forall x \in A'$. Based on this, we can define the variable binding of a triple pattern as follows:

Definition 5.7 (Variable binding of a triple pattern against an RDF instance) *Let ϕ be a match of a triple pattern Q against an RDF instance I_R. Then the* variable binding *of Q against I_R corresponding to ϕ is the function $\phi|_V$, where V is the set of variables in Q.*

We now provide the formal semantics of an XRQ query:

Definition 5.8 (XRQ Semantics) *Let Q be an XRQ query, V be its set of variables, and $\langle v_1, v_2, \dots, v_n \rangle$ the head variables of Q. Let $I = (I_X, I_R)$ be an XR instance.*

A variable binding *f of Q against I is a function over V, such that for every tree (resp., triple) pattern $P \in Q$ whose variables we denote V_P, $V_P \subseteq V$, $f|_{V_P}$ is a variable binding of P against I_X (resp., I_R).*

The result of Q over I, *denoted $Q(I)$, is the set of tuples:*

$$\{\langle f(v_1), f(v_2), ..., f(v_n) \rangle \mid f \text{ is a variable binding of } Q \text{ into } I\}$$

In case of a boolean query, $\langle\rangle$ is true and the empty set of tuples is false.

The definition combines in an intuitive fashion the notion of variable bindings in the RDF and XML sub-instances. When a variable is shared by a tree pattern and a triple pattern, our XQR semantics ensure that it is bound to the URI of an XML node in I_X, and such that some triple in I_R also mentions this URI.

Example. Applying the XRQ query of Figure 2 to the data instance of Figure 1 yields the following result: $\{$"acme@example.com", $\langle$Alice$\rangle\}$. Figure 3 shows the match found for each tree/triple pattern and the variable binding for the entire XRQ query.

Joins and type casting The XRQ language allows one to attach the same variable, say $V, to the property of a triple pattern (which must be a URI, thus belong to $\mathcal{U}$), and the value of an XML node (which is a literal, and thus belongs to $\mathcal{L}$). Rather than considering this a type error in the query, we take the permissive approach of converting all variable bindings to $\mathcal{L}$ and comparing their string representations.

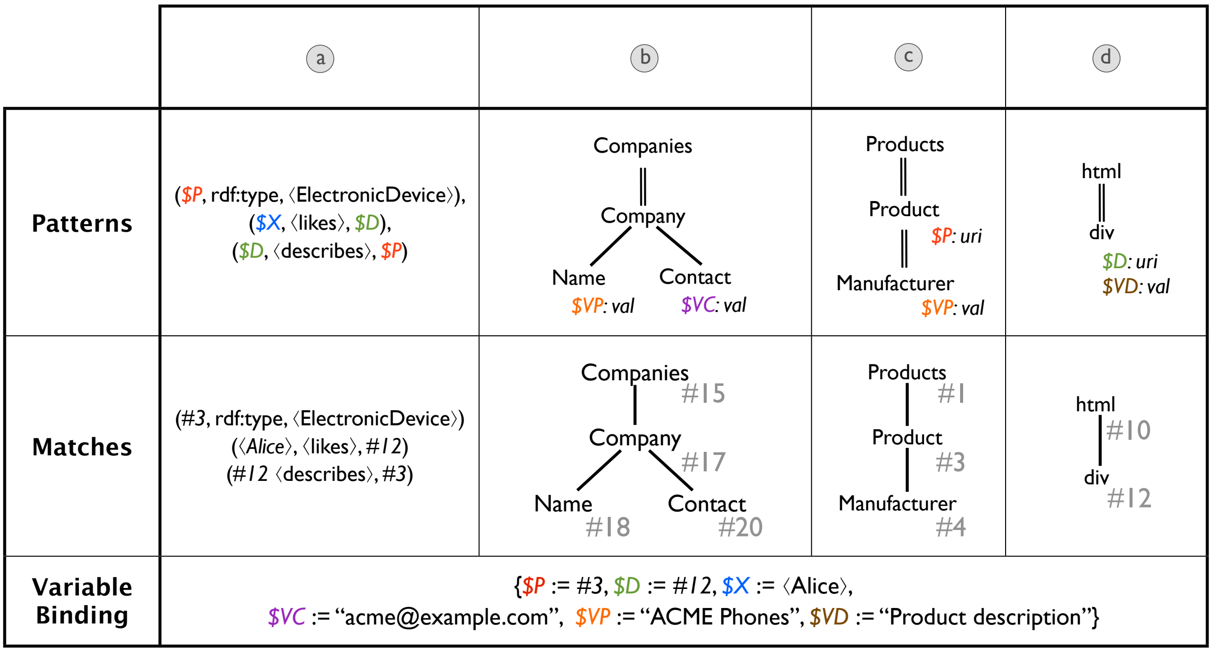


Figure 3: Pattern matches and variable bindings of the query of Figure 2 against the XR instance of Figure 1

5.3 Extended XRQ

As described above, applying an XRQ query returns a set of tuples. However, since the input is an XR instance, one should ideally be able to also create such an instance as the output of the query. To this end, we extend our query language by augmenting it with a constructor. The constructor not only allows the generation of fresh trees and triples in the output but also allows fresh triples to annotate fresh nodes. The definition and the semantics of the extended language are presented below.

Definition 5.9 (Extended XRQ Query) *An* extended *XRQ query consists of the body of an XRQ query and a head of the form* (S_X, S_R)*, where* S_X *is a set of* XML tree templates *and* S_R *is a set of triples. Let* V_B *be the set of variables in the body.*

In each tree $t_x \in S_X$*, internal nodes have* $\mathcal{N}$ *labels, while leaf labels are either (i) from* $\mathcal{N} \cup \mathcal{L}$ *or (ii) a variable* $v \in V_B$ *or (iii) of the form* tree(u)*, for some variable of type* uri $u \in V_B$*. A node* $n_x \in t_x$ *may moreover be annotated with a fresh variable (not appearing anywhere else in* S_X*)* u_x *of type* uri*. Let* V_H *be the set of fresh variables of type* uri *introduced in the query head, and* $V = V_H \cup V_B$ *be the set of all query variables.*

Each triple $t_R \in S_R$ *may use* V *variables in subject, property and/or object position.*

Example. Figure 4 shows an example of an extended XRQ query. Its body is an extension of the query in Figure 2; instead of retrieving the users who like a certain product, it retrieves the e-mails of people they know. Of particular interest is the head of the query. In the bottom part of the head, a new tree links together the maker of a product with the e-mails of friends of users who like the product. In the top part of the query creates new triples that link (through a blank node) the newly created product node with its description and the person who likes it.

We now formalize the semantics illustrated through the example. Let Q be an extended XRQ query, (S_X, S_R) its head, V_B the set of variables in its body and V_H the

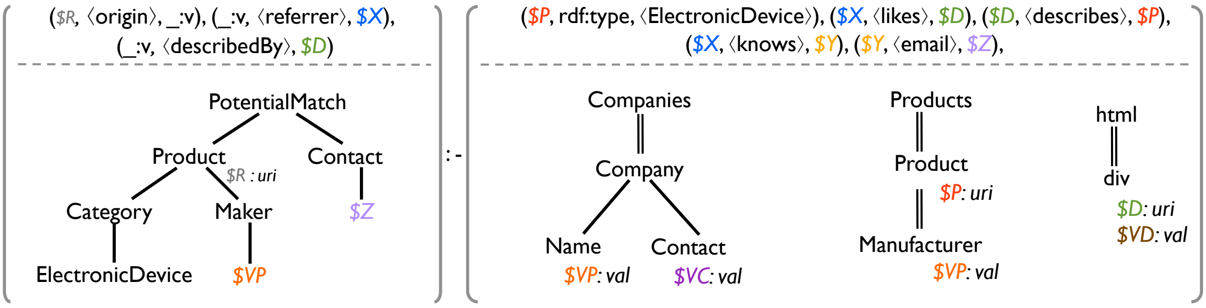


Figure 4: Sample extended XRQ query

set of fresh variables in its head. A variable binding f of Q against an XR instance I is defined as for XRQ queries (Definition 5.8).

Definition 5.10 (Extended XRQ Semantics) *The result of an extended XRQ query Q over $I = (I_X, I_R)$ is an XR instance (I'_X, I'_R) obtained by enumerating all variable bindings of Q against I and, for each such binding f:*

1. Construct I'_X: *For each tree $t_x \in S_X$: (a) If t_x consists of a single node labeled $tree(u)$, for some variable of type* uri *$u \in V_B$, add to I'_X the I node whose URI is $f(u)$. (b) Otherwise, add to I'_X a new XML tree T_X, built by copying t_x and replacing (b′) each t_x leaf labeled with some $v \in V_B$ by $f(v)$, and (b″) each t_x leaf labeled with $tree(u)$ for some* uri *variable $u \in V_B$, with a fresh copy of the I XML node whose URI is $f(u)$.*

   Case (a) above outputs nodes from I_X, with their URIs unchanged.

   Nodes produced in case (b) are assigned new URIs by the URI assignment function f_{uri}. For each $t_x \in S_X$, node $n \in t_x$ annotated with a fresh uri *variable u, and binding f building an XML tree T_X from t_x, the variable u is bound (by the I'_X construction) to the URI of the T_X node built out of n as in (b), (b') or (b'') above.*

2. Construct I'_R: *Copy S_R, replacing each variable v by $f(v)$ if $v \in V_B$, and by the new node URI to which v is bound, if $v \in V_H$.*

The above semantics deserves several comments.

First, notice that I'_X must be built *before* I'_R. This is so that the URIs of the possible new XML nodes of I'_X are known by the time I'_R is built, and can appear in its triples.

Second, I'_X construction is quite complex since the language allows returning nodes from the input I_X, unchanged (case (a)), but also alternatively new XML trees built by stitching together XML trees from I_X in case (b), where query variable bindings both over I_X and I_R can be glued in the returned trees. This expressive power is closely inspired from XQuery return clauses, which can output both existing and new (re-combined) trees. We extend it to include also data from RDF triples.

Finally, the syntactic construct $tree(v)$ was introduced in Definition 5.9 and used in Definition 5.10 to refer to the *tree whose root node URI is the binding of v*. XQuery does not need this, since it binds variables to *nodes*. The price for uniformly working with XML trees and RDF triples in XR is that XR variables are bound to node *URIs*, making this indirection level necessary.

The extended XRQ is closed with respect to the data model, i.e. the result of an extended XRQ query on an XR instance is guaranteed to also be an XR instance (which

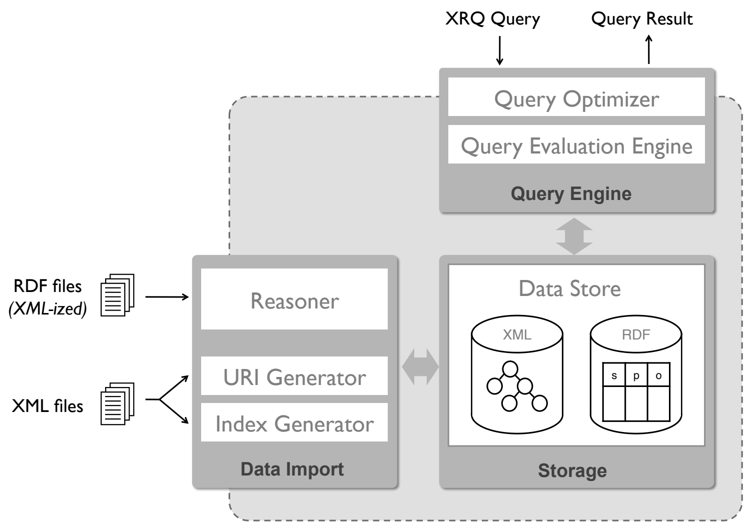


Figure 5: XRP platform architecture

facilitates query composition). In the future, we plan to further enhance XRQ, to allow among others nesting and grouping of the query results.

6 XRP Data Management Platform

We implemented XRP, an XR data management platform prototype, using Java 1.6. XRP supports the storage of XR instances, and the evaluation of XRQ queries; it partially reuses, and extends, the ViP2P platform[5]. Figure 5 depicts the system's architecture. Next, we describe each component in detail.

Data store An XR instance is internally stored in tables (collections of tuples), hosted within a native store using the BerkeleyDB persistent library [8]. RDF triples are mapped to a simple three-attribute relation, as advocated in state-of-the-art RDF data management systems [23]. An XML document d is stored in a tabular form as follows: the database administrator specifies a few tree patterns $t_1^d, t_2^d, \ldots, t_k^d$ *which, together, are sufficient to store all the data from* d. Each of these tree patterns is evaluated over d using an extension of the algorithm in [10], and for each t_i^d, $1 \leq i \leq k$, we store the collection of tuples $t_i^d(d)$ in BerkeleyDB. The tree pattern t_i^d is stored also as metadata describing the tuple collection $t_i^d(d)$. This storage solution is document-specific and requires some human intervention. However, it generalizes easily: many of the existing storage models for XML into relations can be described by some tree patterns, which can be automatically computed from a document or its DTD [5], eliminating the need for human intervention.

We end by noting that XRP implements a wrapper/mediator model, which makes it easy to plug physical data sources of diverse implementations. For instance, in our previous work [6], on which XRP is partially based, an RDBMS stored part of an XML database, while our native store hosted the rest. Thus, one could easily include within XRP more efficient specialized stores built for XML or RDF, while preserving the capability to answer queries over both.

The data store also hosts *indexes*, within our BerkeleyDB native tuple store. As usual, an administrator can manually request the generation of an index that she might find

[5]http://vip2p.saclay.inria.fr

helpful. However, XRP also automatically generates indexes that we have found to be beneficial for most XRQ queries (see "Automatic indexing" below).

Query evaluation engine To evaluate queries, we used and enhanced ViP2P's library of physical operators, implementing the iterator model. Typical ViP2P operators are: scan of a stored tuple collection, selection, projection and join operators. Join operators are of particular interest. The current implementation features HashJoin (implementing an in-memory hash join), and BindJoin, which is a sideways information passing join (also known as a dependent or functional join) [14]. Our implementation of BindJoin iterates over all tuples coming from its left-hand child operator, extracts the values of the join attributes from each tuple, and makes a call to its right-hand child, asking for all its tuples that match the join attribute values. Typically (and in particular in XRP) the main usage of a BindJoin is to join with an indexed data collection by passing to it values for the index key. Access to the index is modeled using a BindAccess operator; more details on these operators can be found in [22].

Query optimizer A query optimizer takes into account the available indexes, cardinality and size statistics etc. and generates a query evaluation plan that is then passed to the query engine for execution. Currently, the optimizer enumerates all possible left-deep trees, and pushes down selections and projections.

RDF saturation Recall from Section 5.2 that XRQ query semantics are defined not on the RDF instance, but on its saturation, which extends the RDF instance with a set of entailed triples. To comply with this, XRP saturates the RDF instance *upon loading*, through a reasoning module borrowed from our work on view selection for RDF databases [17, 16], which implements the RDF entailment procedure defined in the standard [31]. This is a one-time task performed when the RDF data is loaded. Saturation was chosen in this work for simplicity; its efficiency compared to other techniques (taking into consideration the cost of maintaining the database as well) is examined in a separate work [18].

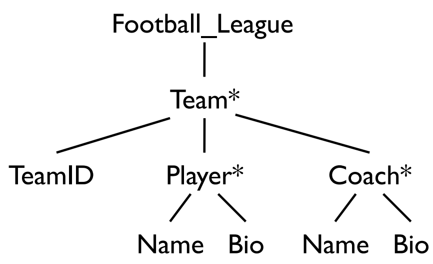
URI generation for XML nodes As discussed in the definition of the XR model in Section 4, URIs are central for XR: they form the "glue" between the XML and RDF sub-instances, allowing RDF triples to refer to XML nodes. To this end, whenever an XML document is imported into XRP, all its nodes should be assigned unique URIs. To generate a node URI, we proceed in two steps: we assign a unique ID to every XML document that is imported into the system; then we assign a unique ID to every node within a document. The URI of a node is the concatenation of the document URI and the node ID within the document. To assign node IDs, many of the existing XML node labeling schemes could be used; ViP2P implements (pre-order, post-order, depth) identifiers [21] which allow inferring, by comparing two node IDs, whether a node is a parent or ancestor of another. ViP2P also supports Dynamic Dewey IDs [36], which, unlike the previous scheme, adapt gracefully to updates. For XRP, we used without loss of generality the (pre-order, post-order, depth) ID scheme.

Automatic indexing By experimenting with XRP, we were able to make useful observations about the queries that are usually executed in the system. In particular, most queries on annotated documents join triples either on their subject or their object with URIs of XML nodes. For instance, the query of our running example in Figure 2 falls under this category. Given the expected frequency of such queries, we decided to have XRP automatically build indexes to speed up evaluation.

This could be accomplished by indexing either the RDF or the XML sub-instance. From the perspective of the RDF sub-instance, triples may be joined (with XML node

```
<!DOCTYPE   Football_League [
<!ELEMENT   Football_League (Team*)>
<!ELEMENT   Team (Player*, Coach*) >
<!ELEMENT   Player (Name, Bio) >
<!ELEMENT   Coach (Name, Bio) >
<!ELEMENT   Name (# PCDATA) >
<!ELEMENT   Bio (# PCDATA) >
<!ATTLIST   Team TeamID CDATA
            # REQUIRED >] >
```

(a) DTD



(b) Graphical representation

Figure 6: DTD of synthetic XML data used in the experiments

URIs) either on their subject or on their object. Thus, indexing the RDF data would entail building two separate indexes (one for the subjects and another for the objects). Since typical XR queries join XML nodes on their URIs, we opted for a single index on the URIs of XML nodes. This index is automatically created for each XML document imported into the system and then subsequently leveraged by the query optimizer when possible. Our experimental evaluation (Section 7) shows that these indexes drastically improve the performance of most queries.

7 Experimental Evaluation

In the following, we study XRP's performance for different queries and data sets utilizing the various processing strategies supported by the system.

7.1 Experimental Setup

Data sets We used a synthetic data set containing information about football teams and players annotated with various properties. The XML sub-instance conforms to the DTD outlined in Figure 6. The documents were generated using the toXgene XML generator [7]. The RDF instance consists exclusively of annotations about players included in the XML data. We generated a total of 12 data sets, whose characteristics are depicted in Figure 7. Each data point on the graph corresponds to a different data set. We varied two independent dimensions: the number of players, which we kept identical to the number of annotations, is shown on the vertical axis, and the size of the Bio elements of each player, which affected the size of the XML file, is shown on the horizontal axis. Increasing the number of players also led to an increase of the RDF file size. Due to this correlation, the latter is also shown on the vertical axis. The annotations were distributed following a random distribution over the players. There is an average of one annotation per player but some players may have more than one, while others may have no annotation at all. The XML data were transformed into relational form by matching a tree pattern identical to the DTD of Figure 6 against the XML document and generating a single tuple per match, as described in Section 6.

Queries We used a set of 3 queries shown in Figure 8. Q_1 looks for ids of teams with some player annotated by a particular property. Q_2 is a generalization of Q_1 looking for teams with players annotated by *any* property. Finally, Q_3 returns properties of players of a specific team. On our data set, queries Q_1 and Q_3 are highly selective (i.e., they return very few results), whereas query Q_2 has low selectivity.

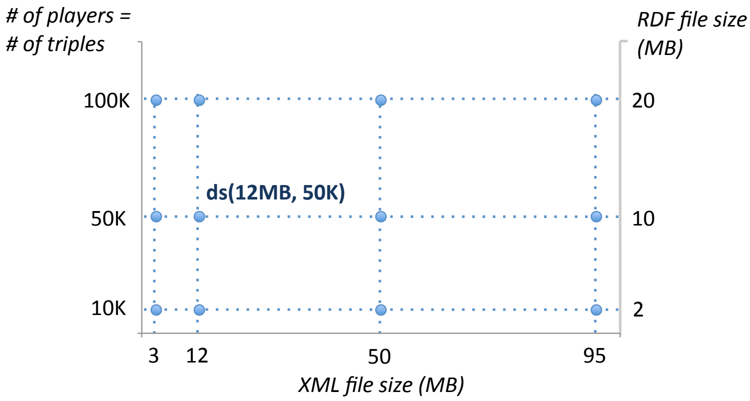


Figure 7: Characteristics of the XR data instances used in the experiments

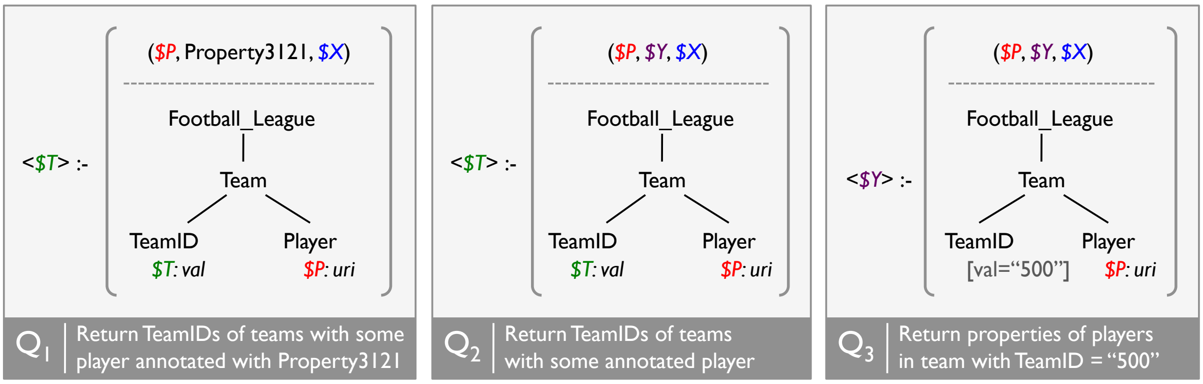


Figure 8: Queries used in the experiments

Execution strategies Each query looks for XML data and *corresponding* RDF annotations. Thus, it contains a tree pattern (for querying the XML sub-instance) and a triple-pattern (for the RDF sub-instance) that are joined. Obviously, a simple way to evaluate the queries is to evaluate the tree pattern query, evaluate the triple pattern query, and then join them in order to identify the annotated elements. Since the join compares URIs (an XML node URI with the URI appearing as the subject of a triple), this join is a simple equi-join and can be executed using the HashJoin operator.

We argue that queries of this form are not an exception but the rule. Indeed most queries on annotated XML documents are expected to join the URIs of XML nodes with RDF triples. To improve the performance of such queries, we looked at materializing indexes and using them at query evaluation time. We experimented with different indexes both on the RDF and the XML data. For the purpose of our experiments, we turned off the automatic index generator of XRP and hand-picked a set of simple indexes which were materialized as follows:

- Index I_1 stores all the team IDs indexed by the URIs of the players in those teams. This is an index over the *XML sub-instance*. We employ this index for Q_1 and Q_2.

- Index I_2 stores all the RDF triples indexed by their subjects. The index is over the *RDF sub-instance* and we use it for Q_3.

Whenever an index is available, one can evaluate the query by accessing the index with the right search keys. In this case, the query evaluation plan is a BindJoin operator over a Scan and a BindAccess operator (which is an operator for accessing an index).

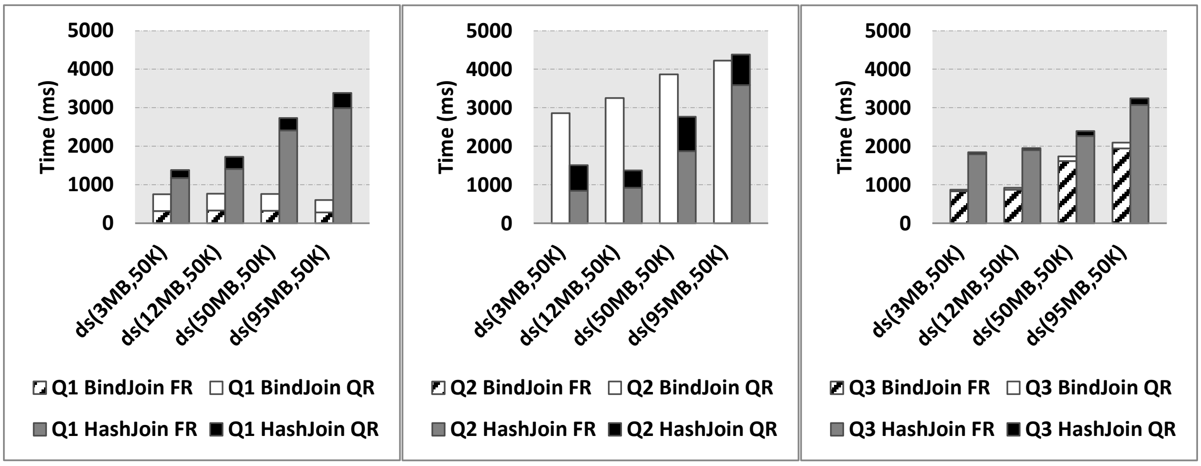


Figure 9: Query Response (QR) and First Result (FR) times for Q_1, Q_2, Q_3 on data sets of varying XML file size

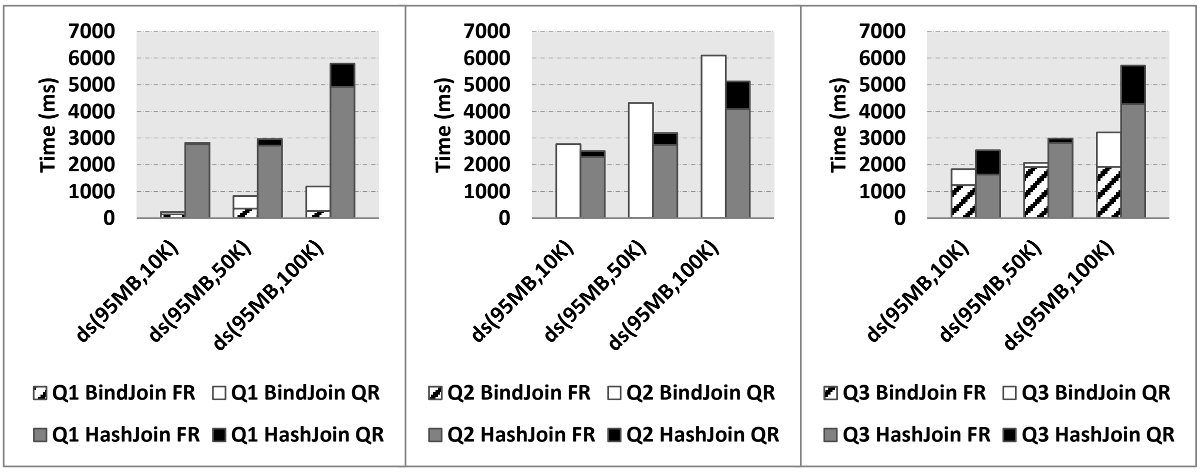


Figure 10: Query Response (QR) and First Result (FR) times for Q_1, Q_2, Q_3 on data sets of varying number of players (and triples)

Observe that for Q_1 the RDF part (i.e., the triple pattern) is the most selective, while the indexes are built on the XML sub-instance; for Q_3, the converse holds, i.e., the XML part is the most selective while the index is built over the RDF sub-instance.

We have run the 3 queries over the 12 data sets by disabling the query optimizer and using in each case manually two strategies: simple (using a HashJoin) and index-based (using a BindJoin). Figures 9 and 10 depict the resulting running times for the most informative subset of the 36 executions. For each data set, query and strategy, we show both the total response time (QR) and the time for retrieving the first result (FR).

Hardware The experiments were conducted on a 2.40GHz Intel Xeon X3430 machine with 8MB L2 Cache and 4GB RAM running Mandriva Linux 2010 with kernel 2.6.31.14.

7.2 Experimental Results

Figures 9 and 10 show the query evaluation times for all three queries over two different subsets of our data sets. In Figure 9, all four datasets have the same number of players and triples, but the physical size of the XML file varies. On the other hand in Figure 10, we fix the XML file size at 95MB and vary the number of players (and triples). In the label of a data set $ds(x, y)$, x indicates the size of the XML file, while y stands for the number of players (and the number of triples), as shown in Figure 7.

Scalability The results indicate that XRP scales well both with the number of nodes in the XML sub-instance (and the number of triples in the RDF sub-instance), as well as with the physical size of each instance. Moreover, the system performs well even for moderately big instances. For instance, for an XML file of 95MB and 100K triples (data set $ds(95MB, 100K)$) and a low selectivity query (Q_2), XRP's total response time (QR) stays below 6.5 seconds.

Effect of indexes on total evaluation time The use of indexes generally results in a speedup. In particular, employing BindJoin instead of HashJoin decreases the total query execution time by a significant factor (up to 4 in some cases). The magnitude of the speedup depends on the ratio between the size of the indexed file to the size of the non-indexed one; the bigger the ratio, the bigger the percentage of time that is saved from the relation scans by utilizing the index and thus the bigger the speedup. For instance, consider queries Q_1 and Q_3 ran on a data set with a 10MB RDF file and a 95MB XML file (dataset $ds(95MB, 100K)$ in Figure 9). Q_1 uses an index on the largest file (i.e., the XML) and thus the difference in the performance between the BindJoin and the HashJoin is much larger than for Q_3, which uses an index on the smaller file.

The only query where using BindJoin actually decreases performance is Q_2. The reason is that Q_2 does not involve any selection neither on the XML sub-instance, nor on the RDF sub-instance. Thus neither side of the join operator is more selective than the other (in particular, the join operator has to join 50K RDF triples with 50K tuples from the XML instance for the data sets in Figure 9). This causes as many index lookups as the number of tuples in the non-indexed file, and results in comparable performances between BindJoin and HashJoin, which scans the non-indexed file. Moreover, for small XML files BindJoin performs actually worse than HashJoin. This is due to the overhead incurred from accessing all items of an index in a random order, compared to sequentially scanning a relation with the same data items.

As the XML file size increases though, the difference between the performance of BindJoin and HashJoin decreases. The explanation is simple: BindJoin depends mostly on the number of index accesses, which in turn depend on the number of triples in the RDF file. Since these stay the same as the XML file size increases, BindJoin scales sub-linearly w.r.t. the latter. On the other hand, HashJoin is significantly affected by the XML file size. If however we keep the file size constant and increase the number of tuples, BindJoin becomes progressively slower as shown in the graph for Q_2 in Figure 10.

Effect of indexes on time to first result However, it is interesting to note that BindJoin can still be of great benefit when considering the time to the first result (FR). Since we can utilize the index to access the indexed side of the join, getting the first result requires simply scanning the other side of the join operator until the first match (that also satisfies all selection conditions of the query). This may lead to very fast FR times, especially when the first match is close to the beginning of the file. For example, in the case of Q_2 in Figure 9, FR times are close to zero, because the first match is among the first triples in the RDF file. On the other hand, the same times for Q_3 in Figure 10 are much closer to the QR times, because the players of team "500" (selected by the query) are further down in the file. These observations can be of great importance especially for real-time applications that require a fast turnaround time.

8 Conclusion

The need for efficient ways of creating and querying annotated documents is becoming increasingly apparent. In this work we make a first step in this direction by formalizing a data model for representing such documents, a query language for seamlessly querying them according to both their structure and annotations and a prototype implementation demonstrating the feasibility of our approach. As part of our future work, we plan to extend the expressivity of our system by enhancing the query language with additional constructs (e.g. grouping and sorting) and also to study additional optimizations that may be possible when XML and RDF data are stored within the same system.

References

[1] S. Abiteboul, T. Allard, P. Chatalic, G. Gardarin, A. Ghitescu, F. Goasdoué, I. Manolescu, B. Nguyen, M. Ouazara, A. Somani, N. Travers, G. Vasile, and S. Zoupanos. WebContent: Efficient P2P Warehousing of Web Data. *PVLDB*, 2008.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] W. Akhtar, J. Kopecký, and T. Krennwallner. XSPARQL: Traveling between the XML and RDF worlds - and avoiding the XSLT pilgrimage. In *ESWC*, 2008.

[4] S. Amer-Yahia, S. Cho, L. V. Lakshmanan, and D. Srivastava. Minimization of Tree Pattern Queries. In *SIGMOD*, 2001.

[5] A. Arion, V. Benzaken, and I. Manolescu. XML access modules: Towards physical data independence in XML databases. In *XIME-P*, 2005.

[6] A. Arion, V. Benzaken, I. Manolescu, and R. Vijay. ULoad: Choosing the Right Storage for Your XML Application. In *VLDB*, 2005.

[7] D. Barbosa, A. O. Mendelzon, J. Keenleyside, and K. A. Lyons. ToXgene: A template-based data generator for XML. In *SIGMOD*, 2002.

[8] Oracle Berkeley DB Java Edition. http://www.oracle.com/technetwork/database/ berkeleydb/overview/.

[9] S. Bischof, S. Decker, T. Krennwallner, N. Lopes, and A. Polleres. Mapping Between RDF and XML with XSPARQL. Technical report, DERI, 2011.

[10] Y. Chen, S. B. Davidson, and Y. Zheng. An Efficient XPath Query Processor for XML Streams. In *ICDE*, 2006.

[11] O. Corby, L. Kefi Khelif, H. Cherfi, F. Gandon, and K. Khelif. Querying the Semantic Web of Data using SPARQL, RDF and XML. Research report, INRIA, 2009.

[12] S. Dill, N. Eiron, D. Gibson, D. Gruhl, R. Guha, A. Jhingran, T. Kanungo, S. Rajagopalan, A. Tomkins, J. A. Tomlin, and J. Y. Zien. SemTag and seeker: bootstrapping the semantic web via automated semantic annotation. In *WWW*, 2003.

[13] M. Droop, M. Flarer, J. Groppe, S. Groppe, V. Linnemann, J. Pinggera, F. Santner, M. Schier, F. Schöpf, H. Staffler, and S. Zugal. Bringing the XML and Semantic Web Worlds Closer: Transforming XML into RDF and Embedding XPath into SPARQL. In *Enterprise Information Systems*. Springer Berlin Heidelberg, 2009.

[14] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of binding patterns. In *SIGMOD*, 1999.

[15] T. Furche, F. Bry, and O. Bolzer. Marriages of Convenience: Triples and Graphs, RDF and XML in Web Querying. In *Principles and Practice of Semantic Web Reasoning*. Springer Berlin / Heidelberg, 2005.

[16] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. RDFViewS: a storage tuning wizard for RDF applications (demo). In *CIKM*, 2010.

[17] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *PVLDB*, 5(1), 2012.

[18] F. Goasdoué, I. Manolescu, and A. Roatis. Saturation-based versus reformulation-based query answering in RDF databases. Submitted for publication, 2011.

[19] S. Handschuh and S. Staab. Authoring and annotation of web pages in CREAM. In *WWW*, 2002.

[20] K. Karanasos and S. Zoupanos. Viewing a world of annotations through annovip. In *ICDE*, pages 1133–1136, 2010.

[21] Q. Li and B. Moon. Indexing and Querying XML Data for Regular Path Expressions. In *VLDB*, 2001.

[22] I. Manolescu, L. Bouganim, F. Fabret, and E. Simon. Efficient Querying of Distributed Resources in Mediator Systems. In *CoopIS*, 2002.

[23] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *VLDB J.*, 19(1):91–113, 2010.

[24] P. Patel-Schneider and J. Siméon. The Yin/Yang web: XML syntax and RDF semantics. In *WWW*, 2002.

[25] L. Reeve and H. Han. Survey of semantic annotation platforms. In *ACM SAC*, 2005.

[26] J. Robie, L. M. Garshol, S. Newcomb, M. Biezunski, M. Fuchs, L. Miller, D. Brickley, V. Christophides, and G. Karvounarakis. The syntactic web. *Markup Lang.*, September 2001.

[27] M. Vargas-Vera, E. Motta, J. Domingue, M. Lanzoni, A. Stutt, and F. Ciravegna. MnM: Ontology Driven Semi-automatic and Automatic Support for Semantic Markup. In *EKAW*, 2002.

[28] RDF in HTML. http://research.talis.com/2005/erdf/wiki/Main/RdfInHtml, 2006.

[29] GRDDL. http://www.w3.org/TR/grddl/, 2008.

[30] Microformats. http://microformats.org/.

[31] RDF. http://www.w3.org/RDF/, 2004.

[32] RDF concepts and abstract syntax. http://www.w3.org/TR/rdf-concepts/, 2004.

[33] RDF Semantics. http://www.w3.org/TR/rdf-mt/, 2004.

[34] RDFa Primer. http://www.w3.org/TR/xhtml-rdfa-primer/, 2004.

[35] URIs, URLs, and URNs: Clarifications and Recommendations 1.0. http://www.w3.org/TR/2001/NOTE-uri-clarification-20010921/, 2001.

[36] L. Xu, T. W. Ling, H. Wu, and Z. Bao. DDE: from dewey to a fully dynamic XML labeling scheme. In *SIGMOD*, 2009.

[37] K.-P. Yee. CritLink: Advanced Hyperlinks Enable Public Annotation on the Web. In *Computer Supported Cooperative Work (CSCW)*, 2002.